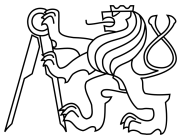
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY



ASSIGNMENT OF BACHELOR'S THESIS

Title:	Numerical database system
Student:	Miroslav Mašat
Supervisor:	Ing. Ivan Šimeček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

1) Study and explore the current concept of numerical database system (see [1]) that stores most frequent search terms in a weighted search tree.
2) Describe its limitations, constraints, and efficiency.
3) Discuss possibilities to expand the current implementation (focus on a parallel execution).
4) Perform a performance comparison of your implementation with the original one on a public dataset (see [2]).

References

[1] S. C. Parkb, a, C. Bahria, J. P. Draayerb, a and S. -Q. Zhengb: Numerical database system based on a weighted search tree, Computer Physics Communications, Volume 82, Issues 2-3, September 1994, Pages 247-264
[2] Google Inc. (2015). Women's World Cup Players. Retrieved from https://github.com/GoogleTrends/data/blob/gh-pages/20150512_WomensWorldCupPlayers.csv

L.S.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 4, 2016

Czech Technical University in Prague
Faculty of Information Technology
Department of Theoretical Computer Science

Bachelor's thesis

Numerical database system

Miroslav Mašat

Supervisor: Ing. Ivan Šimeček, Ph.D.

16th May 2016

Acknowledgements

I would like to thank my professor Ing. Ivan Šimeček, Ph.D. for all the help during the creation of this thesis by always giving vast and valuable feedback. Also, thanks to my family and friends for tremendous support during all my study at Czech Technical University.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 16th May 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Mašat, Miroslav. *Numerical database system.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Cílem této práce je nastudovat a implementovat algoritmus numerického databázového systému s využitím datové struktury ohodnoceného binárního stromu. Implementace v jazyce C++ bude srovnána s původní implementací v jazyce Fortran. Obsahem bakalářské práce je analýza existujícího řešení, jeho parametrů a diskuze možností implementace paralelní varianty tohoto algoritmu.

Klíčová slova databázový systém, numerický systém, binární strom, ohodnocený strom, vyvážený binarní vyhledávací strom, vyhledávání v stromové struktuře, paralelizace algoritmu, implementace databázového systému, C++

Abstract

The objective of this thesis is to conduct a study and implement the numerical database system using a weighted binary tree data structure. The implementation in the language C++ will be compared to the original implementation in Fortran programming language. The portion of this thesis is dedicated to the analysis of the current solution, its parameters, and discussion of implementing the parallel variant of the algorithm.

Keywords database system, numerical system, binary tree, weighted tree, balanced binary search tree, search in the tree structure, parallel execution of an algorithm, implementation of the database system, C++

Contents

List of Figures

Introduction

Database

Many times in a practical development of systems or applications, there is a need for a database. A database is a collection of information. [2] This collection can be organized and can contain multiple types of objects, often raw values of data. We can imagine the database as a table, that have multiple rows (columns) that represent key and optionally multiple columns (rows) that represent values. An example of database represented as a simple table can be found below in the figure 0.1. Usually, we consider key to be unique and distinct but that is not compulsory. Data stored in database should reflect reality and structure to support processes and manipulation of information.

School trip atendees		
Number	Name	Age
0	Anna	16
1	Beth	17
2	Bob	16
3	Clara	18
.	...	...
9	Tom	16

Figure 0.1: Example of database in table layout

For instance, in the database provided above (0.1), key is the number of the attendee. This key is of ordinal type (integer), and all keys are distinct. We can also say, that we index two values per each key, being the name of the attendee and their age respectively, in this example. Additionally, we can see that the size of the database is equal to a number of distinct keys and it is 10, in our case, since there are 10 kids attending the school trip. This

database of students going for a trip, nicely summarises all details, and is a perfect example why and how are databases useful.

Database Management System (DBMS)

DBMS stands for "Database Management System." In short,a DBMS is a database program [A.1]. Differently put, when translating a concept of a database into software engineering reality and applications, we talk about database management system [3]. At this point, we already talk about a software program that interacts with either directly the user or other software programs, in order to store or retrieve information from the database. There are many DBMS already, serving different purposes. Every of these systems can tackle the implementation of the information management differently. DBMS creates an interface of a kind. So, while a database is the single concept of organized information, DBMS is a very concrete software solution to define and manage this organization.

There are many conventional DBMS that addresses most cases and needs of reliable, fast or scalable storage. Examples of well known DBMS are:

- MySQL
- PostgreSQL
- Microsoft SQL Server
- Oracle
- Sybase
- SAP HANA
- IBM DB2

Although these are different software, made by different business entities, and are generally not of the same implementation, systems share similarities to allow an exchange of information between some.

Numerical Database system

Although there are many DBMS mentioned earlier, there are occasions, where a need of an unconventional or tailored solution is needed. The solution, that possess different qualities, parameters or general principle. There are certain fields, where, the fast operations or the space effectiveness are desired. On the other hand, there are tradeoffs that need to be accepted, in order to ensure

desired parameters and properties. Another speciality of database systems as such, is the data interpretation, use of the main data structure. Last but not the least, some databases are friendly to easy to use the parallel resource of machines and some are not. Depending on the requirements and resource this is also an aspect when choosing the database system.

Numerical database system is a special database system that addresses a very narrow problem of storing a limited amount of data based on a popularity of certain elements [1]. This means, that this database system will favor elements that are retrieved often and/or very recently. Keeping recent or often used keys in the system can also help in many scientific calculations, where often or recent calculations can be reused later on. Fast operations of search and insert are desired. To make the system fast, setting the maximal capacity of the system is required.

I have chosen this topic in order to bring more insight and popularity into this topic. There is currently not an extensive use of a system like this. These systems could be an important part of the future computer systems that require exceptional searching speed and performance at all other costs. In addition, there is expected to be further developments in terms of academic level.

Potential usage of the system

The numerical database system could have many uses. Typical candidate situations to use a solution such as this would be e-commerce software that would allow tracking of the most popular items in the offer. Also, environments that are limited by memory are a great option since this database system can be configured to work well with available memory in mind. For example, many applications could be considered for using this system as a cache for all sorts of data used. Another application can be considered when looking on offline versus online synchronization tools and last but not the least all sorts of science experiments, where calculations or subroutines of many kinds can be used in the very near future while the program is still running.

Overview of the thesis

There are three chapters in this thesis explaining the basic concept of the system, implementation specifics, comparison to implementations that are already available and a discussion of parallel enhancements to the algorithm. In chapter one, there will be theoretical background (literature overview), asymptotic notations and principle of the algorithm. After basic requirements and expectations, data structures explanation is provided. Multiple structures are discussed and evolved into final picture, that will be realised in the second

chapter. There is the brief discussion about the state and possibility to make the algorithm parallel. Finally, the flowchart of the main search routine is provided, to explain all mentioned in a way of a diagram.

The second chapter, realisation, will first, explain why the C++ as a language was chosen with respect to algorithm specifics and test data. Rest of the chapter will be dedicated to efforts of actually implementing the algorithm. Main interfaces and code samples will be provided, to show how the system is implemented.

The third chapter will be just briefly comparing both older documented, and my solution. There will be two parts to this comparison, where the first part will touch on the implementation differences. Different technologies, code paradigms and data structures will be compared to show, what are the changes and how these contribute to a better system. Second part will be dedicated to performance graphs of my solution and comparison to the original performance.

In the conclusion, I will be talking about overall use and give a recommendation based on expected usage of a solution as such. I summarize how succesfull this thesis was and what future developments might bring.

Chapter 1

Theoretical Background

1.1 Basic principle

As mentioned previously, the numerical database system will be able to store a limited number of keys. Keys are of primitive ordinal type (for example integers, floats, characters or strings), so for given 2 keys it is possible to determine which of two is larger (smaller). In our case we will consider keys to be distinct, hence, the case of two keys being equal is irrelevant.
Numerical database system that will be described in this thesis will consist of **two** main **components**:

- **Data component** - will store keys alone, and hence will provide purely storage for data. This component will facilitate operations naturally expected from any database system such as search, insert, and delete.
- **Priority component** - will help to keep another important parameter of the system, which is, storing only relevant data - entries we use often or recently. For this purpose, we also restrict the maximal capacity, to set the largest possible number of keys being kept. This additional component also helps to determine, which element should be removed from the database system in order to allow for more recent or relevant keys.

These components will both store keys, but each component for a different purpose. This is integral, both these components need to coexist and all operations over the database system will interact with both at the same time. In the later part of theoretical background chapter, we will be looking at both these components and try to propose adequate data structure. To each of those, there will be a discussion of more options to choose the right one, that will be concretized and implemented in the chapter: realisation.

Finally, to describe the functionality, it was mentioned we will expect our system to be able to perform **three** basic **operations**:

- **find** (search) **operation** will retrieve the key requested, if present in the database system
- **insert operation** will enlist the new key into the database system if capacity allows
- **delete operation** will free up space by removing the key from the database system

This description wraps the principle, basic structure, supported operations and hence describe the system all in all.

1.2 Asymptotic notations

To set the expectations, we will provide asymptotic notations [4] of respective operations. In the following figure, there are all three supported operations over the numerical database system [1] with asymptotic notations provided, see figure 1.1. These are the complexities we will be trying to achieve.

Asymptotic notations of numerical database system	
Operation	$\mathcal{O}$-average
find (search)	$\mathcal{O}(\log n)$
insert	$\mathcal{O}(\log n)$
delete	$\mathcal{O}(\log n)$

Figure 1.1: List of average and worst asymptotic notations

1.3 Data component

For analyzing and recommending the right data structure for the numerical database system, we will start with data component, with the objective of exclusively storing and retrieving keys alone with no respect to capacity or priority. Given the asymptotic notations above (see 1.1), we need a data structure that allows for operations mentioned and tune it into performance desired.

1.3.1 Binary tree data structure

For a start, I will be describing binary tree data (BT) structure and its properties, together with the explanation of benefits and reasons why this particular data structure would be used. An example of the BT can be seen below. See figure 1.2. At the beginning, it is important to understand a basic BT. BT is a one case of K-nary trees, where k=2. Every node of the tree has a parent node

(with an exception of a root node) and can have 0, 1 or 2 child nodes. We call these nodes left and right children, although this is only considered to be a convention. [5] The connection within parent and child nodes is illustrated by oriented vertices going from the parent to the child node. Hence the graph with oriented vertices we call BT to be a directed graph. BT has no cycles. We consider 3 basic types of nodes:

- Node with 0 children - the leaf
- Node with no parent - the root node
- Inner node - all nodes contained in the tree that are not a root node nor leaves

Every node has a key that is in this example of the numeric type (integer), but by a definition can be of any ordinal type. In figures provided, the key is presented inside of a circle of a corresponding node.

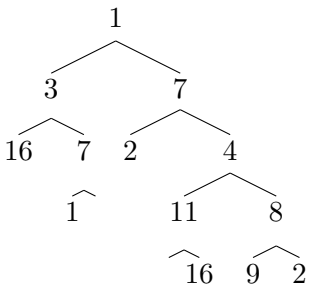


Figure 1.2: Example of Binary Tree

1.3.2 Binary search tree data structure

In this algorithm, in addition to using BT, it is required to be a binary search tree (BST), which is a special case of a BT. The special property of Binary Search Tree (BST) is, that keys of nodes in the tree (in our case integer values) are ordered, forming a representation of values that are sorted.

This representation is again described conventionally as ascending (so that the previous element of the tree is smaller or equal to the next element) [6]. Nodes are also placed logically in a way, that for every node, its left child has always a smaller (smaller or equal) value than the node itself while its right child has always a larger or equal (larger) value than the node itself. All these properties can be again seen and reviewed. See figure 1.3.

Keeping this structure is required. One of the parameters of the numerical database system is, to be indexed in the way, that the search operation takes

the minimum time with the respect to asymptotic notations table mentioned earlier. The result of a search is determined by a key given. In the case of a numerical database, this is an integer value , but by a definition can be of any ordinal type. To gain and keep this property, it is needed to be able to quickly determine where to look, for a certain key given.

After both requirements of key being ordinal and data been kept ordered (which is achieved by the careful creation of the BST from scratch), we can apply a variety of binary search algorithm, effectively performing the search on the set.

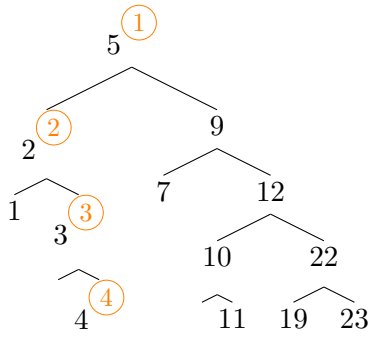


Figure 1.3: Example of Binary Search Tree

Binary search, working on a tree such as shown in the figure 1.3 works in a way of comparing a current searched key with a reference entry. The algorithm starts at the root, hence, a root is the first current searched key. If the reference entry is larger (smaller) than the current key, then we set the new current key to the right (left) child of the old current key. This way we are recursively searching the tree entering inner nodes (which are in fact subtrees), level by level until hitting the level of leaves. The search ends the moment the current searched key is equal to reference entry (and we claim the entry found) or if there is the last leaf already explored, there are no more children of the current node (the reference entry was not found).

To illustrate the search process in the example tree provided in figure 1.3, when searching for a key $k = 4$, there is an order of visiting each node starting at the root and finishing at the node with a key that have been searched for. Order of visiting nodes of the BST (1.3) is displayed by an each node (order o as an integer value in a orange circle: (o) above the key of a node).

This tree does not have all properties to be structured to use effectivity of binary search and if the tree is not balanced, the complexity of search can reach up to $\mathcal{O}(n)$, [6] which can not be considered as fast search by any means,

but in the average case the result is much faster and can get close to $\mathcal{O}(\log n)$ [1]. Similarly, the operation of insert and delete depend on the concrete tree and hence the asymptotic complexity can vary. The example of the tree that is a valid BST but is not optimal is displayed below, see figure 1.4.

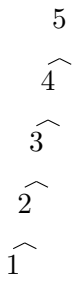


Figure 1.4: Example of problematic Binary Search Tree

As seen on the tree above (1.4), this tree is binary and fulfills all the criteria that BST has. We call this tree unbalanced. The unbalanced tree has one or more nodes with an uneven number of children. By randomly adding elements and being unlucky, this situation can happen. In this case, the insertion into the tree did go in the following order:

- insert(5)
- insert(4)
- insert(3)
- insert(2)
- insert(1)

Where searching a tree like this for a key $k = 1$, the number of steps to go through this structure (maximal depth of the tree) is the same as searching in a one-way linked list (which it basically converted to) and takes n steps in the worst case.

1.3.3 balanced Binary search tree data structure

To amend problems with unlucky inserts and so, in order to ensure $\mathcal{O}(\log n)$ complexity at all times, while searching, we have to make tree balanced. The example of a balanced tree can be seen in the following figure: 1.5. The difference to a tree displayed in the figure: 1.3 can be seen from a point of relative symmetry of left and right subtrees. This symmetry can be described also by describing the tree being balanced.

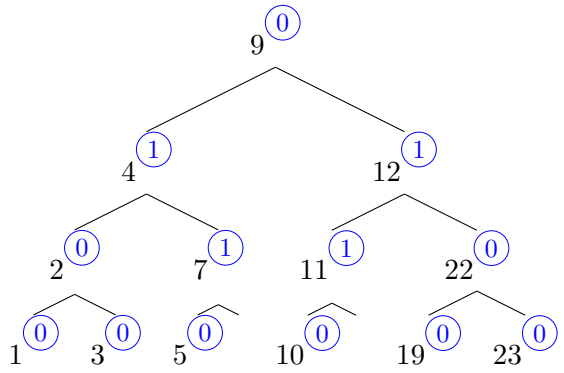


Figure 1.5: Example of balanced Binary Search Tree

There are multiple tree structures that are balanced but differ in design or implementation. Some of the notable structures that keep balanced effectively are:

- 2-3 tree
- AA tree
- AVL tree
- Red-black tree
- Scapegoat tree
- Splay tree
- Treap

Many of these structures are somehow unfamiliar or not suitable for this system. For example, the 2-3 tree does not work on a binary basis, but rather ternary and also, two keys in a key-key pair are stored in the each node [7]. This would not satisfy the way we want to search the tree later on. While some structures are not suitable, other are too complicated to use in a parallel manner, such as Splay tree, where even accessing an element causes a rotation that is called splaying. [8] That also complicates the implementation. In some, the asymptotic complexity can also vary depending on the data inserted (Splay tree) [8], or being randomly influenced (Treap) [9].

From within all these structures, the most sensible ones are

- AVL tree
- Red-black tree

Both these trees are balanced, easy to implement and widely used. By keeping in mind our objective of being able to search in the database as fast as we can, we pick between the two the one more balanced. AVL tree is more rigidly balanced than red–black tree. If we look at the operation of find (search), while both of these trees have an average asymptotic complexity of $\mathcal{O}(\log n)$ [10], the real cost lies in the maximal depth of trees. See the table below (1.6)

Average and maximal depth of different trees.		
Tree Name	average depth	maximal depth
AVL tree [11]	log$n$	$1.44 \cdot \log(n)$
Red-Black tree [12]	log$n$	$2 \cdot \log(n+1)$

Figure 1.6: AVL versus Red-black tree in terms of tree depth

On the other hand, if we look at the management of balance, this is done less frequently at Red-Black trees compared to AVL tree structure. All other operations have the same average complexity, but at this point we prefer the find operation to any other one, the AVL tree will be implemented.

By having a height-balanced AVL tree we can fully leverage all pros and performance of binary search which is ensuring $\mathcal{O}(\log n)$ complexity of search option by design [10]. Hence, every node needs to fulfil the condition of a balance factor. This has to be kept also regardless of operations performed over the database and at all times. Let's define the depth of the node. The depth of a node is the number of edges from the node to the tree's root node [13]. If we take all the nodes of a tree (subtree) and we find a maximal depth, that is considered a height of a tree (subtree).

Let $\mathcal{H}l$ ($\mathcal{H}r$) be the height of a left (right) subtree respectively to a node n. Then we define balancing index b as $b = abs(\mathcal{H}l - \mathcal{H}r)$. If $b < 2$ then we can consider a tree (subtree) balanced. It is also a convention to keep balancing factor for an each node, together with information about its key and data. For better understanding all following figures: 1.7, 1.8, 1.9 and 1.5, balance factor is displayed by an each node (balance b as an integer value in a blue circle: $\textcircled{b}$ above the key of a node). There are 3 cases for b [1]:

1. $\mathcal{H}l - \mathcal{H}r = -1$: the height of right subtree is larger than the height of a left subtree
2. $\mathcal{H}l - \mathcal{H}r = 1$: the height of left subtree is larger than the height of a right subtree
3. $\mathcal{H}l - \mathcal{H}r = 0$: the height of both left and right subtrees are the same

The criterion of height-balanced subtrees can be damaged by an insertion of new nodes by a conventional method of appending a new entry as a leaf to the existing tree. Similarly, a break of balance can be caused by deletion of a node. After such an incident, it is desired to correct the tree and bring it back into the balanced state. Algorithms of so, called rotations, can be used, see figures 1.7 and 1.9. These cases are invalid since right (left) subtree of node C (A) has the smaller height than left (right) subtree by more than 1. After an application of rotation on 1.7 (1.9), the balanced state is achieved: 1.8. Additionally, there are two more cases, that need to be addressed. These are displayed in figures 1.10 and 1.11. In this situation, two subsequent rotation operations are needed to achieve the balanced state: 1.8.

If these operations are called after every insertion (deletion), we say the tree is self-balancing, also being referred to as AVL self-balancing tree (Georgy Adelson-Velsky and Evgenii Landis' tree, named after the inventors). All these rotations performed over a self-balancing tree have $\mathcal{O}(1)$ complexity.

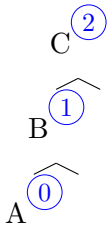


Figure 1.7: Unbalanced Tree (Left-Left)

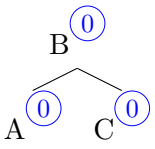


Figure 1.8: Balanced tree

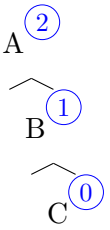


Figure 1.9: Unbalanced Tree (Right-Right)

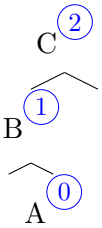


Figure 1.10: Unbalanced Tree (Left-Right)

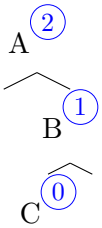


Figure 1.11: Unbalanced Tree (Right-Left)

1.3.4 Weighted Binary search tree data structure

Apart from keeping AVL balanced tree, the property of algorithm is to keep only entries that are somewhat important or recent. The determining the importance of a node comes from multiple factors. The first factor is the source

data itself. In the data source, each and every entry (node) has already defined an initial priority. Let priority p, be a positive, nonzero value. Example of tree with priorities assigned to an every node (priority p as an integer value in a red circle: (p) above the key of a node), see the figure: 1.12. This means, that when the tree is initially constructed, each node already has a priority assigned. At the beginning, the tree is filled with first *cap* nodes, where *cap* is a maximal capacity of the free.

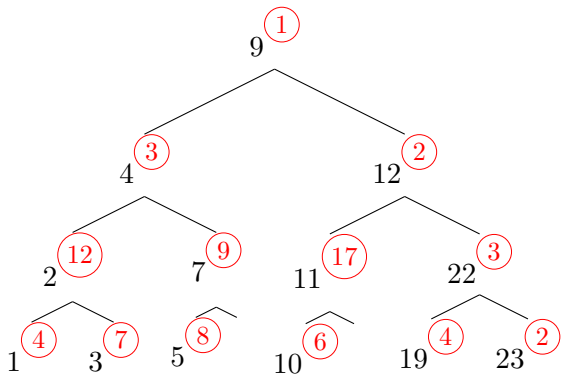


Figure 1.12: Example of Weighted Binary search tree

This is important since the program relies on the fact, that when the entry is searched for, there is a probability, it will be searched for it again. The algorithm will naturally first go through the tree, trying to find the entry requested after it fails to do so, it will search through items not present in the tree. If the record is found outside of the structure of the tree, it will be added to the tree or replaced one or more records with the lowest priority. For a simple case, we will assume, that every entry takes equal memory storage in the tree.

Let this concrete example (1.12) to be a data component of the numerical database system having the capacity $c = 13$ and contain 13 keys already. Then, by inserting a new node n with a key $z = 8$ and priority $p = 12$, the node with the smallest priority (which happens to be a root node r of the whole tree with a key $z = 9$ and priority $p = 1$, in this case) is being removed. After removing the root node, the whole tree needs to be rebuilt and a new node needs to be inserted at the correct position. The newly created tree is displayed for a reference, see: 1.13

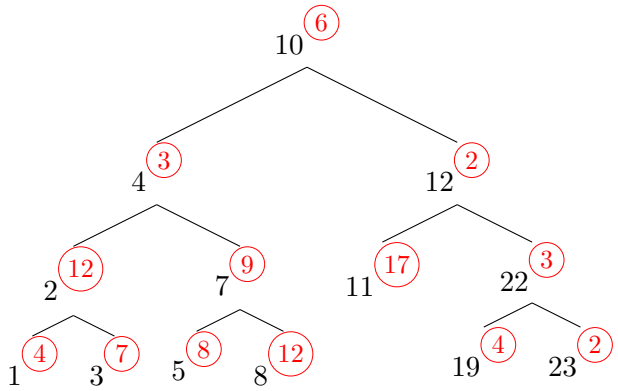


Figure 1.13: Example of Weighted Binary search tree

After each insertion or deletion, we need to check the balance factor and perform one or the series of rotations accordingly, to keep the tree a valid AVL tree.

1.4 Priority component

The limited capacity of the system brings interesting situations we need to solve.

As mentioned earlier, entries that are searched for often or recently should be favored over other ones. While the data component is not full, adding any element is as simple as appending it to data component (AVL tree) and do a rotation if needed, as explained in the last section.

Another case is, when the entry that is sought or inserted, is already in the structure of the tree. In a case like that, after a retrieval of the entry, the priority of the node is incremented by a fixed or variable value. This can vary by implementation and intended use, more on this will be provided in the realisation chapter.

As seen on the insertion procedure, once there is insufficient space for a new entry to be propagated inside the tree, the algorithm has to quickly pick up the node with the smallest priority and remove it from a tree making the space for a new node. These are some of the scenarios where keeping only the data component would not help to decide which entries to keep and which not to.

To sum up this brief intro, there is a need to keep this priority component separate and define how it will be used.

Priority component should:

- Guard the capacity of the database system and be able to yield the number of currently inserted keys in data component

- Provide the key with least priority to inform the system of least important entry, passing it for deletion.

Priority component will store all of the keys as much as these are stored in the data component. In our case additionally, each node will be assigned and updated a priority property, that will consist of a non-negative float number (we can use any ordinal data type, such as integers, strings or characters).

In this section we will consider, what approach and different data structures can be used based on the preference of either convenience or the performance, storing these priorities. There are many options how to store priorities of nodes in the tree. Basically, any data structure with a possibility to store multiple values can be used. Some of the options include, but are not limited to:

- Unordered linked list (queue)

- Ordered linked list (priority queue)

- Binary heap

It is again a matter of preference of convenience over speed or vice versa. The queue (priority queue) is able to perform the task of enqueue (dequeue of max/min) in the constant time of complexity $\mathcal{O}(1)$. But the operation of dequeue of max/min (enqueue) at queue (priority queue) has a much worse complexity of $\mathcal{O}(n)$. Hence, the worst case complexity of selected operations (enqueue/dequeue) can get as worse as $\mathcal{O}(n)$ in the queue (priority queue), and this would be a major bottleneck of the whole priority component. [6]

1.4.1 Binary heap data structure

While linked lists are not suitable for the task of keeping the most (least) prioritised node, an ideal solution is provided by the max (min) heap. An example of the (max) binary heap can be seen in the following figure: 1.14. A binary heap is a tree structure itself having certain properties. A binary heap is a tree that keeps the maximal (minimal) element at the top of the tree, being a root node, we say that the heap is hence also max-heap (min-heap). Also, this tree is always a complete binary tree [14], by the definition.

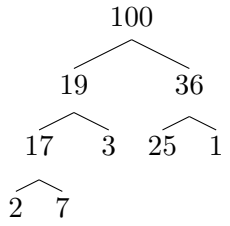


Figure 1.14: Example of binary (max) heap

Given the example figure (1.14) it is apparent, that an every parent of the max (min) heap is larger (smaller) than the maximum (minimum) of its children. This property of orienting larger (smaller) nodes of max (min) heap towards the top of the tree (root node) is being maintained at all times and together with the completeness of the tree that is storing the heap. This relationship between parent and children nodes is ensuring, that the maximal (minimal) key of the collection is, in fact, bubbled up and settled as the root of the max (min) binary heap [15]. After any operation, except searching, performed over the heap structure, one of the repairing operation shall be performed.

There are two main repairing operations that keep the property of the max (min) binary heap:

- Bubble-up, that needs to be performed at the insertion of a new key.
- Bubble-down, that needs to be performed at the removing of a key

Bubble down reparation will start at the newly inserted node and check if a parent node is having a key (in our case priority) that is ordinarily smaller (larger) than the key of the parent itself in the min (max) binary heap. If this is not fulfilled, then switching of child and parent node will be executed (one or multiple times) until the structure does not possess all properties needed. Example of such reparation is displayed at the following figures: 1.15, 1.16, 1.17.

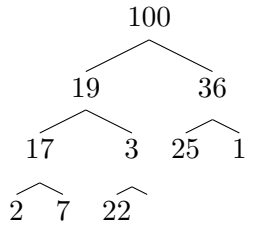


Figure 1.15: Step 1 of rebuilding of a binary (max) heap

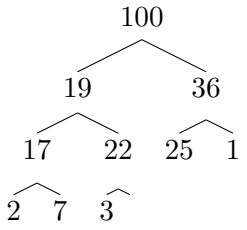


Figure 1.16: Step 2 of rebuilding of a binary (max) heap

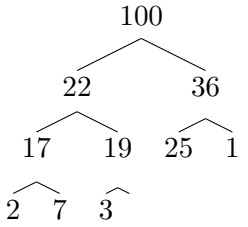


Figure 1.17: Binary (max) heap after a succesfull rebuild

An example of such a process (in a max heap scenario) is displayed in figure 1.15, where the leaf valued 22 is not smaller or equal than its parent valued at 3. Subsequently, after this switch, the problem persists as can be seen in the figure 1.16 only, this time, the problematic node is the one valued 22 in the relationship with its parent 19. Only in the final figure 1.17, we see the proper binary (max) heap that has all sufficient properties desired. This operation of rebuilding the heap is very simple and always requires the maximum of steps given the level of the tree that the new element is inserted into. Hence, the worst case scenario is, that the newly inserted node (which is always a leaf in the tree), will bubble up all the way to a position of the root of the whole tree. Since the tree is binary complete, this operation has a worst case complexity of $\mathcal{O}(\log n)$ by design. [6]

Likely, there is a bubble-down repair needed at some point. This is typically required after an operation of extracting the maximum (minimum), which happens to be a root node of a binary max (min) heap. A Similar process works for any node removed from the heap, but in our case, we will be removing mostly the root node.

The reparation process of the structure is simple and quick. The root of the binary heap (or any node that is being deleted) is replaced by the last previously inserted node (the leaf) and criteria of this new node and its children is checked again for the general property and subsequently switched, in case the property is violated. This process continues recursively until all parents in the binary max (min) heap are of larger (smaller) or equal key (priority) than their both children. Example of an operation of extracting the root of binary (max) heap, followed by the operation of bubble-down can be seen on these figures: 1.18, 1.19, 1.20.

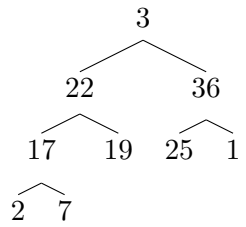


Figure 1.18: Step 1 after root removal

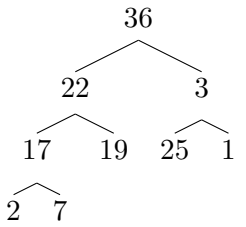


Figure 1.19: Step 2 after root removal

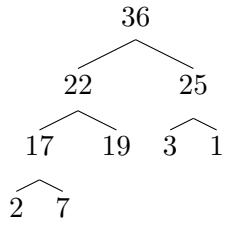


Figure 1.20: Binary (max) heap after a succesfull removal

As can be seen, the first step of removing the root tree (1.18), the binary (max) heap is not in the correct state, since the root node valued 3, is larger than both left and right nodes valued 22 and 36 respectively. This needs to be corrected and so, the larger (smaller) child node is switched with the parent one (1.19). This continues when the similar situation described above populates, so node keyed 3 is a parent of children nodes valued 25 and 1. Only after this switch, the structure is finally repaired to the state that is good and valid, see the final figure 1.20.

This whole procedure of removing the root node can again initiate a process of repair of the structure described above. The average case complexity of the bubble-down action is again only $\mathcal{O}(\log n)$ at worst [6]. Naturally, the algorithm only checks one of the two subtrees every run. This is due to the fact, that, the other subtree is already a valid binary max (min) heap. Hence, a large portion of the structure is always intact and not manipulated with. This also simplifies the implementation.

1.5 Parallel discussion and suggestions

The problem of current implementation [1] of the numerical database system is being unsuitable for concurrent processing. All procedures of inserting, deleting and searching over both data and priority component in the previous implementation are simply not thread safe [A.3]. Example can be seen in the following figures below: 1.21, 1.22. This is due rotations. On these two figures, we can see two threads. Let's assume, that at one moment, the thread T1 is searching for an element with a key of 18, while the T2 thread is trying to remove the root node with the key of 14. This situation is illustrated in the figure 1.21. Let's assume, that the thread T2 is faster and do the deletion first. According to standard deletion routine, the deleted root is replaced by the successor of the node, which, in this case, is the node with a key equal to 18. After removal of this key, the tree is not balanced and hence, the rotation occurs. This situation leads to the figure 1.22.

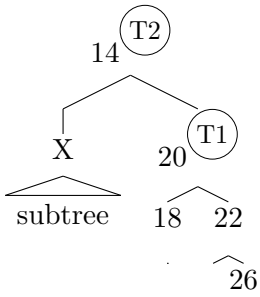


Figure 1.21: relaxed binary tree

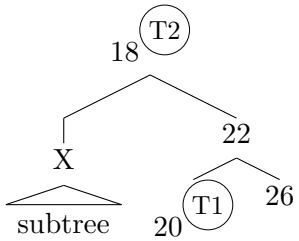


Figure 1.22: relaxed binary tree after root removal

The problem comes when (at 1.22), the thread T1 is still hanging at the node keyed 20 and hence, can not find the node with a key 18. Since the key 20 has no children at the moment, the algorithm can not traverse anymore and yields key 18 not being found, although, the key is indeed still in the tree. This is only one problem, but explains, why with no thread control of the operations flow, locks or different other routines, the operations over the AVL tree will fail.

If a tree would not be self-balanced, the tree structure could not be changed so dramatically and no rotations would take place. In that case, the approach to concurrent processing would be less complicated.

During the research about balanced trees that would allow concurrent processing, I came across two main concepts that are recycled between many studies done up to this day.

- Internal binary trees utilizing quick lookup of predeessor (successor) [16]
- External versioned binary trees utilizing growing (shrinking) rotations [17]

I will very simply show how both of these ways work on the operations of search and try to explain the cases, where rebalance may interfere with the actual retrieve operation, similarly to figures above (1.21, 1.22). It is also important to understand, that actually all of other operations rely heavily on the search routine. Delete and Insert operations are partly a search operation, where firstly the node (position of the node) has to be found in order to delete (insert) it.

1.5.1 Internal binary trees utilizing quick lookup of predecessor (successor)

The first concept utilizes an enriched node structure, which is a building block of the tree structure [16]. Every node does contain not only the key, left and

right child and preferably a parent node, but also a predecessor and successor of the node. Another change is that the nodes, that are deleted are only deleted logically, so the node also has a flag of being invalid [16]. This interpretation has some advantages and disadvantages. Advantages being, that the implementation of search capability is very straightforward and, that delete operation is in fact simplified, by keeping the invalid node in the tree and only releasing it in the case of having only one child [16]. The disadvantage of such setup is, that the predecessor and successor have to be always up-to-date for every node per every operation, since, the tree search depends on these pointers. Example of handling the problematic case from previous section can be seen, looking at previous figures 1.21 and 1.22.

Similarly, while the T1 thread would be for a search of the node keyed 18, this node would already be populated as the root node of the tree. The search does not end here for this concept, because, when reaching the leaf node, the find mechanism will try to traverse by predecessors and later successors and try to isolate the candidate for a search match [16]. if unsuccessful, then the node is definitely not found and the search is over. In this case, while thread T1 would reach the end of the tree and hang there, a simple lookup on its predecessor would discover the node keyed 18, even though now, this node is the root node.

This mechanism is super simple and intuitive, more details are provided in the separate paper.

1.5.2 External versioned binary trees utilizing growing (shrinking)

The second concept does rely on the tree being partially external [17]. The external binary tree does only consider valid keys that are stored as leaves. All other nodes (internal nodes, including the root node) are called the routing nodes [17] [16]. This representation has an advantage as well as disadvantage. The advantage is being able to delete nodes easily, simply by removing them, since every key is stored as a leaf. The disadvantage is, of course, the waste of space, since routing nodes are only taking up space with no contribution to operations performed [17]. This tree is also utilizing the idea of growing/shrinking of a node and using this as a measure of thread regulation and retry routines. Example of such tree and how growing/shrinking is used is displayed below in figures:

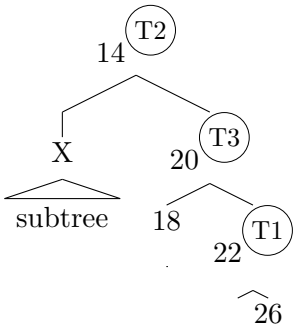


Figure 1.23: relaxed binary tree

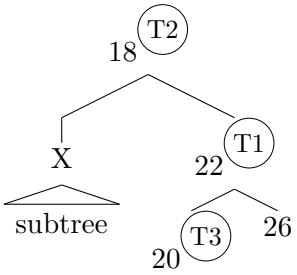


Figure 1.24: relaxed binary tree after root removal

On figures above, we can see the same operation that caused the problem before, but in this case, I will explain how this external tree handles such situation. This tree keeps an information about nodes present in rebalancing actions. In this case, nodes 22 and 20 were rotated due to the removal of the root node keyed 14. This algorithm relies on the premise of growing/shrinking node [17]. The growing (shrinking) of the node does realise, when the depth of the node is shortened (lengthened). Hence, in this case, we say, that the node keyed 22 had grown while the node keyed 20 had shrunk. The good thing about growing nodes is, that we do not have to do anything, as their position did not change prominently enough to break the searching routine. The shrinking operation did, though. In case we witness shrinking of a node, the handling of versions is introduced and retry routines are executed in order to fix this problem [17].

This approach is definitely much more complicated, but performance-wise adds the most boost to systems with many cores/processors.

1.5.3 Summary of parallel analysis

The numerical database system can be run concurrently if both data and priority components are parallel-friendly. As shown in this section, there are some viable working concepts of such modifications on binary trees. Recommendation hence the simplicity and the practicality goes to an internal binary tree utilizing quick lookup of predecessor (successor) [16].

1.6 Conclusion of theoretical background

As mentioned before, for the numerical database system 2 main components are needed. After this theoretical background, consideration of asymptotic notation and multiple refinements of data structures, these structures were chosen:

- **AVL balanced tree** - for **data component** of the numerical database system

- **Binary minimal heap** - for **priority component** of the numerical database system

These two structures will be realised in the next chapter according to pseudocode, to construct this numerical database system. Parallel enhancements were proposed, but the implementation of such solution is out of the scope of this thesis.

1.7 Flowchart of the algorithm

Below there is a flowchart diagram of the algorithm, the flowchart does not include the construction step of the database, only the process of the lookup for an entry in the system. Beware, that search functionality, in fact, includes remove and insert subroutines. See the figure 1.25.

The whole database si more complex than this diagram and is using this only as the main component that loops for each entry that is being retrieved. Starting and finishing points are marked as red squares while blue diamond shape describes the process of acquisition of the searched term. Orange squares are highlighting different processes and in the code, these will be implemented as functions or procedures, and green diamond shapes describe logic components of choice, which in the code are implemented as if or $switch$ statements.

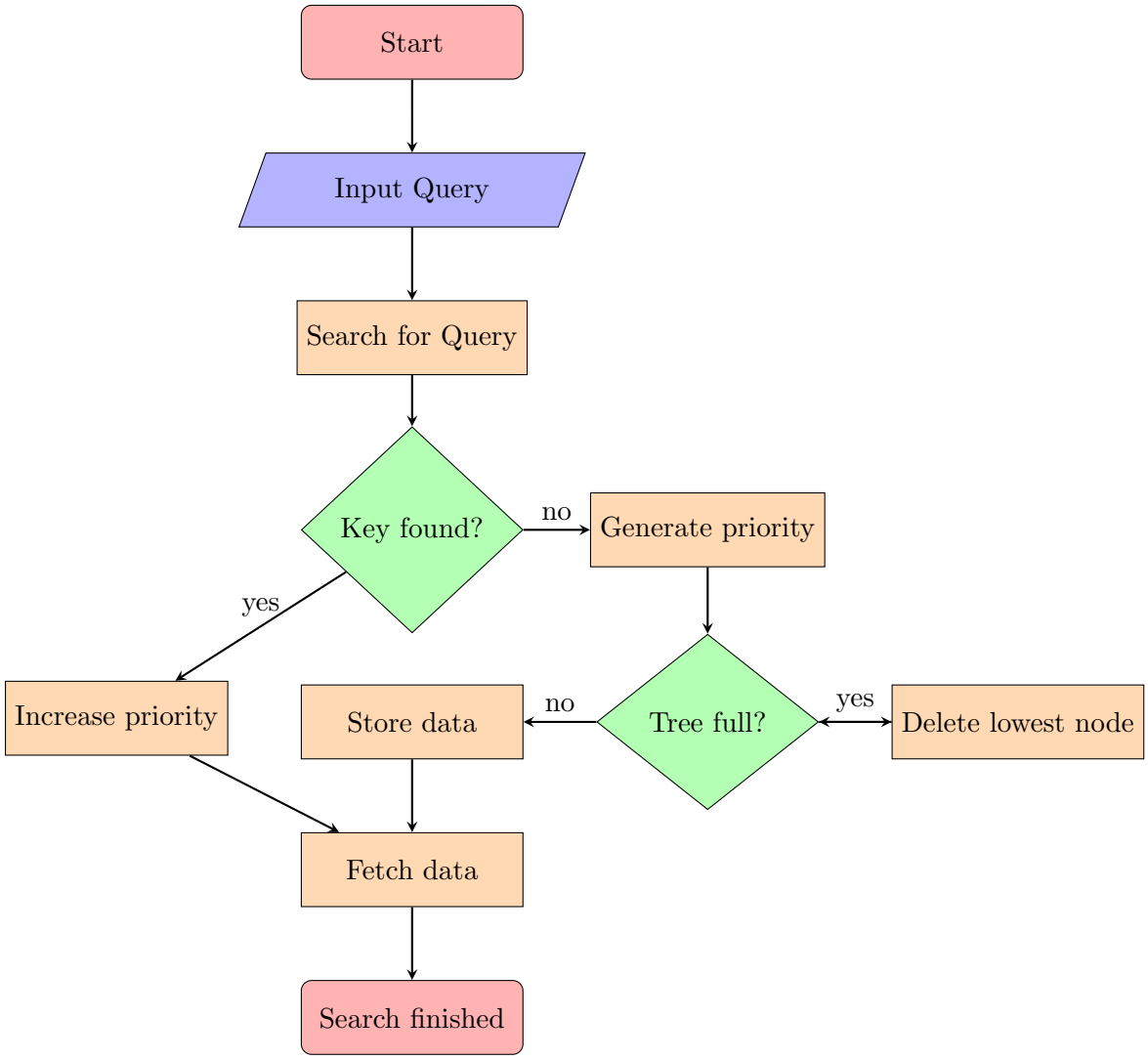


Figure 1.25: Flowchart diagram of a search functionality of the database system

The process displayed in the flowchart is the only main component of the database system, this search functionality is also having a functionality of adding/removing element from it. It is important to understand this diagram as this one process displayed in figure 1.25 is presenting the whole concept that will be described and realised in the next chapters, so all data structures, functions, and procedures will be contributing to the creation of this process.

Chapter 2

Realisation

In this chapter, I will be discussing the concrete implementation of the algorithm with data processed in mind.

2.1 Programming language selection

For implementing numerical database system, I have chosen C++, being the programming language of the choice. C++ is a general-purpose programming language. It provides instruments of working with memory on a low level while providing object-oriented capabilities. C++ was targeted for use on embedded systems and system programming, with constrained resources. C++ is cherished for its performance, flexibility and efficiency too. Apart from embedded systems, C++ is widely used for software infrastructure for many solutions, including DBMS (mostly SQL) or industries, such as e-commerce or web search. [18]

As the most compelling reason to use C++ as the language for this database system is, how portable and supported this language is, by all main architectures and operating systems. There are also multiple compilers (most known being GCC and LLVM) being able to transform the code into the executable file.

2.2 Requirements of data

Ultimately, this database system should offer a system for management of information that it will be tested on. Data that will be the system for are in this case search results of Google Inc. search engine, with a weight aggregated over the time period. [19]

In this dataset, the key is of an ordinal data type - string, and this string associates with a name of the player. Player names usually consist of a number of words, that are separated by a space. The dataset is not perfect and sometimes, only one of the names of the player is provided, which does not affect the purpose and informational value, though.

The dataset is containing a several hundred names (507 to be exact) of players with a respective priority assigned. Priority provided is of the float number data type.

The realisation will be catered to satisfy inputs from this source, although will also allow for different ordinal data type as keys of the entries.

2.3 Architecture and structure of the program

As C++ language is object-oriented, the whole coding style will be tailored to use the most of this feature. That being said, the program will be logically split and divided into several classes and structures, that allow for using not only as the complete database system but also as individual parts. Implementation will hence allow to modify or reuse data and priority component separately, at the wish of the developer, in future developments.

The implementation will also allow for all basic ordinal data types (integers, floats, doubles, chars, strings, etc,) as being keys. This is achieved by using a built-in template system provided by C++ language. This allows a dynamic setting of supported data type being the key at the moment of initialization. This makes the system more versatile and allow for different use in the future.

The program will be logically divided into these custom classes and structures:

- CTerm - a term carried and stored in the database
- CTermTree (utilises CTerm) - a binary search tree structure that contains all search terms
- CPriority - priority sign of search terms
- CPriorityHeap (utilises CPriority) - binary minimal heap that contains and manages nodes of priority sign
- CDatabase (utilises both CTermTree and CPriorityHeap)

With respect to previously component oriented infrastructure, these classes fall under following components:

- **Data component** consists of CTerm structure and CTermTree classes
- **Priority component** consists of CPriority structure and CPriority-Heap class

CDatabase class will only facilitate as the wrapper for both components, to make the system more wholesome. Besides all these classes, there is a test file, that creates the instance of the CDatabase and loads data into the structure, performs some operations to test validity and shows the overall way of how to use the numerical database system.

For full overview of files, please see the and contents of the medium enclosed, see attached directory tree in appendix D.

2.4 C++ class descriptions and basic notations

In this section, I will explain all five main classes of the system, explaining what concrete methods each class has, constructors, destructors and what data types are used for public or private properties. This should be only very brief, for full implementation, please check with the source code in the medium attached.

2.4.1 Data component: CTerm and CTermTree

2.4.1.1 CTerm class, explained

First off, I will explain the concrete implementation of the node, that is contained in data component (AVL tree). Excerpt of the header of this class is provided below, see the figure [C.1] in attachments apendix.

This base of the AVL tree is responsible for keeping a key, of variable type, the integer that signifies the balance factor needed for tracking whether the subtree is balanced, and three pointers to other AVL nodes being: left subtree, right subtree, and parent node. The pointer to parent is needed in order for rotations to work well. The class only has 2 public methods and those are a constructor, to create a node and a destructor to free up memory by deleting the node.

2.4.1.2 CTermTree class, explained

Class CTermTree encapsulates actions around a tree consisting of CTerm nodes. An excerpt of the header of this class is provided below, see the figure 2.1.

```
template <class T>                                                    1
class CTermTree {                                                     2
public:                                                               3
    CTermTree          (void);                                        4
    ~CTermTree         (void);                                        5
                                                                      6
    bool insert        (T key);                                       7
    void deleteKey     (const T key);                                 8
    bool search        (const T delKey);                              9
    void printInOrder();                                              10
    T findMin();                                                      11
                                                                      12
private:                                                              13
    CTerm<T> *root;                                                   14
    CTerm<T>* rotateLeft              ( CTerm<T> *a );                15
    CTerm<T>* rotateRight             ( CTerm<T> *a );                16
    CTerm<T>* rotateLeftThenRight     ( CTerm<T> *n );                17
    CTerm<T>* rotateRightThenLeft     ( CTerm<T> *n );                18
    CTerm<T>* findMin                 ( CTerm<T> *n );                19
    void rebalance                    ( CTerm<T> *n );                20
    void setBalance                   ( CTerm<T> *n );                21
    void printInOrder                 ( CTerm<T> *n );                22
};                                                                    23
```

Figure 2.1: CTermTree.h

This class is constructing and maintaining the AVL tree. The instance of the class consists of root pointer, that exposes the main root of the tree and then couple of methods, that help with basic operations of inserting, searching and deleting a node from the tree, based on the key. There are also a couple of operations keeping the tree balanced, such as setting the balance factor of a node, rebalancing the subtree around the node, and 4 respective rotation functions, that correspond to cases described eairlier in this thesis. Additionally, the class has some supporting functions for finding a minimum of the tree/subtree, constructor to create the node and destructor to free up memory by deleting the tree.

2.4.2 Priority component: CPriority and CPriorityHeap

2.4.2.1 CPriority class, explained

As of the CPriority class, it stores the key together with priority in one minimal structure. CPriority is the one entry of the minimal binary heap (CPriority-Heap), see the figure [C.2] in attachments apendix.

The class mainly stores the variable key and its priority, that is, of data type float. There are only two notable public methods and those are a constructor, that creates a key with an assigned priority for it and a destructor, to free up memory by deleting the key and corresponding priority.

2.4.2.2 CPriorityHeap class, explained

CPriorityHeap is a vector formed of CPriority classes in order of minimal binary heap, to wrap all keys and priorities into one data structure allowing for operations needed, An excerpt of the header of this class is provided below, see the figure 2.2.

```
template <class T>                                          1
class CPriorityHeap {                                       2
public:                                                     3
    CPriorityHeap();                                        4
    ~CPriorityHeap(void);                                   5
                                                            6
    bool    insert(T key, float priority);                  7
    T       deleteKey(const T key);                         8
    T       deleteMin();                                    9
    void    printHeap();                                    10
    T       findMin();                                      11
private:                                                    12
    vector< CPriority<T> > heap;                            13
    float minPriority;                                      14
                                                            15
    bool    bubbleUp(int i);                                16
    bool    bubbleDown(int i);                              17
    int     minChild(int i);                                18
};                                                          19
```

Figure 2.2: CPriorityHeap.h

This class of minimal binary heap of priorities that come with individual keys consist primarily of the vector of priorities of all keys. Additionally, there is a non-negative float value of minimal priority (the priority of first index int the vector). There are several methods allowing for basic operations of insertion, search, and deletion. Apart from those, there are a couple of methods to maintain the proper structure of the minimal binary heap, notable ones being methods for bubble-up and bubble-down of a key, according to the

heap criteria. As in all classes, there is a constructor and destructor, to free up memory by deleting the whole heap structure.

2.4.3 The main component: CDatabase

Finally, CDatabase class just constructs both data and priority component inside one instance, and maintains operations from the global scope, interacting with instances of both individual components. An excerpt of the header of this class is provided below, see the figure 2.3.

```
template <class T>                                          1
class CDatabase {                                           2
public:                                                     3
    CDatabase(int s);                                       4
    ~CDatabase(void);                                       5
                                                            6
    bool     insert    (T key, float priority);             7
    T        deleteKey (const T key);                       8
    T        search    (const T key);                       9
    void     display   ();                                  10
private:                                                    11
    int size;                                               12
    int index;                                              13
    CTermTree<T>       DTree;                               14
    CPriorityHeap<T>   PTree;                               15
};                                                          16
```

Figure 2.3: CDatabase.h

As can be seen from the header file, there are indeed both components (CTermTree and CPriorityHeap) constructed, covering both data and priority components. Both these classes can be initialized with a variable data type of the key, as needed. Besides these two, there are two integer values marking the capacity of the database system (size) and the current number of valid keys, that are in the system (index). Then, there are three basic operations for inserting, searching and deleting keys and a useful display function, that can graphically show the content of the database. Finally, as always, there is a constructor that assembles the object with the maximal capacity provided by the user and destructor function to remove the numerical database system from the memory.

Chapter 3

Performance review and comparison

My numerical database system, that was explained and realised in this thesis was hugely influenced by an algorithm previously designed [1]. The design and my implementation were done with a couple of important changes in mind.

There are 3 main differences, that were done, in order to enhance the system and bring better usage to the overall concept of the database system:

- OOP approach
- Data type (in)dependance
- Data structure use

3.1 OOP approach

Original [1] implementation done before was developed, in the language Fortran, which at the time, was procedure oriented [20]. At that point, Fortran was still widely used. Hence, the code produced has no OOP paradigms, which prevents this implementation from being extended in an easy and convenient way. The implementation of WSTREE [1] is done in one, rather large (more than 1900 lines long) file. The code is hard to navigate and understand and it is impossible to separate this code into multiple standalone parts.

That is why, I have chosen C++ programming language, that fully embraces the OOP approach. In my implementation, I wanted to make sure that both, data component and priority component are functional and independent from each other and from the database system itself. This way it is easy, to use only one of the parts alone, in a different project or, to switch one or both of

the parts. This can help to experiment with the algorithm in future development. The typical use case would be, to change the data component and use RedBlack or Splay tree instead of AVL tree, and measure, the impact of such a change.

3.2 Data type (in)dependance

The numerical database system described in the study before [1], was very narrowly designed for numerical keys. That means, that the user could indeed only insert, find or delete keys, that are of integer data type. This decision has many reasons, one of them is, that naturally, system restricted on integers is faster in comparison to strings or characters. With an integer, there exist far fewer instructions on the machine level which can perform a comparison. [21] In a process of search, which is the most important operation of a database system, many comparisons are needed, while traveling the data component (AVL tree). While comparing integers takes only the one compare operation, character arrays or more sophisticated strings can take as many comparisons as the length of the searched key.

My implementation is using a template system of C++ language and hence gives the database system power to work with any ordinal data type being the key.

3.3 Data structure use

In the initial implementation [1], both data component and priority component were saved as arrays. This comes from the fact, that every binary tree structure can be implemented as an array [22]. In my implementation, I tried to avoid this approach. The reason being is, that working with an array is not thread safe [23]. Thread safe procedure is always logically correct when executed by several threads [A.3]. If we would like to enable these arrays or containers such as STL vector, we would need to introduce a synchronization primitive, while performing read and/or write operations. One of such approach is called Readers–writer lock [A.2]. This would require another data structure besides the simple array still. One of the objectives of this thesis was to alter the algorithm to support concurrent operations. This is only possible when different keys in both data and priority components are independently stored on different memory addresses, linking to each other. Each of this dynamic node structure needs to be protected by some synchronization primitive but does not influence other node structures in a tree.

Another change in structuring data lies in data component. In the previous implementation [1], the node in the AVL tree had only two links. The

article refers to the data structure used as being a multiset data structure [24]. My implementation uses three links, two for children nodes and one for a parent node [10].

3.4 Summary of improvements

To summarize the improvements made, I made the numerical database system to be as **modular** as possible, and due to this requirement used different language (C++) and different programming paradigm (OOP). Also, I tried to make the code more readable by splitting it into more pieces, logically classes. Subsequently, I tried to make the system as **universal** as possible by using template system features of C++ language and allow for any data type to be used as the key. This helps to use the system for many tasks and use cases. Last but not the least I tried to alter the data structures of the implementation done before, by changing the representation of data component into a structure that uses individual nodes, stored individually, instead of arrays and ease the future developments and addition of **concurrent processing** of the operations.

3.5 Performance measurement

To measure the performance of my implementation done in C++, I run some tests against the database system. To verify how the system behaves under different circumstances, I have defined two testing scenarios, that should address majority of use-cases:

- **100% insertions** test - covering adding of elements into not full database system, see figure: 3.2
- **75% insertions** and **25% deletions** - covering mostly adding and some deletions while the database is full, see figure 3.3
- **40% insertions**, **10% deletions** and **50% retrievals** - covering the most realistic functioning of the system, see figure 3.4

Both tests with corresponding graphs are displayed below with each showing two different results of performance according to data types that were used for keys.

- **Strings** - full names of people, randomly generated, long of 9 - 15 characters
- **Integers** - numbers, randomly generated in interval between 10,000 and 99,999

Both these datasets were large enough containing up to 25000 unique entries and so to measure at different sizes and show how the performance of the database can scale up.

3.5.1 Hardware and software enviroment

Hardware and software used for tests described earlier is displayed in the table below (3.1):

Hardware					
CPU	Intel Core i7 2.2GHz	RAM	8GB 1333MHz DDR3	HDD	256GB SSD
Software					
OS	OSX 10.10.5	Arch.	Apple LLVM version 7.0.2	Comp.	g++ -O3

Figure 3.1: Hardware and sofware used to test

As can be seen above (3.1), quite a powerful machine was used. As fo the software used, I am compiling my numerical database system on the architecture of LLVM compiler infrastructure project [A.4]. When compiling, I am using the binary executable g++ with a parameter $-O3$, which is one of the most aggressive sorts of optimizations [25]. For running of the numerical database system from enclosed SD-card media, please see the Makefile attached for particular compilation options. Also, the full hardware report of the machine can be found there.

3.5.2 Test with 100% insertions, 0% deletions

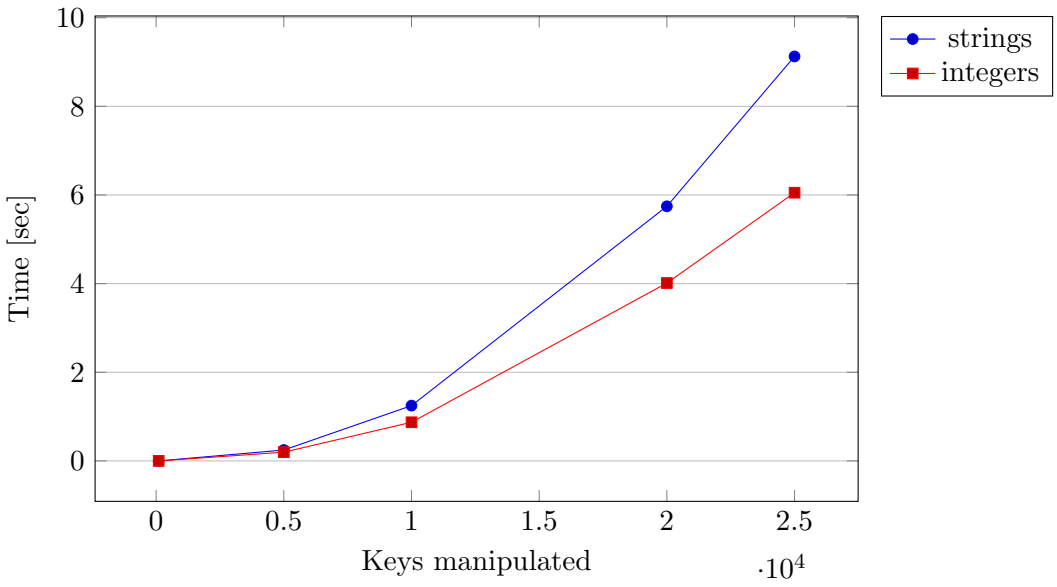


Figure 3.2: Test with 100% insertions, 0% deletions

In the first graph above (3.2), the basic test is shown, to display how inserting routine with no intervention of delete/find routine is employed (the database is yet not full). This is to show, how the time grows linearly in terms of integer keys, while, in the case of string keys, the process slows down a bit, because more and more similar keys are being accumulated and hence, the compare function takes longer while traveling the tree. Also, the observation is, that at since 1000 keys volume, the time difference is more prominent.

When it comes to the data component (AVL tree), the addition of time complexity in both graphs is caused by rotations. It is important to understand that rotations can be populated up high in the tree (in fact, it can rise and affect even the main root node). That said, the deeper the tree is, the more damage rotation routines have. All this is also influenced by the order of elements inserted.

Another consideration that has to be taken into mind, while looking into this graph is the priority component (minimal binary heap), that is all the time employed, while insertions are taking a place. Every inserted element into the system does execute the bubble-up routine, that is quite complex, and can be as demanding as the insertion to the data component itself. This time again, this effect is influenced by the order of keys inserted.

3.5.3 Test with 75% insertions, 25% deletions

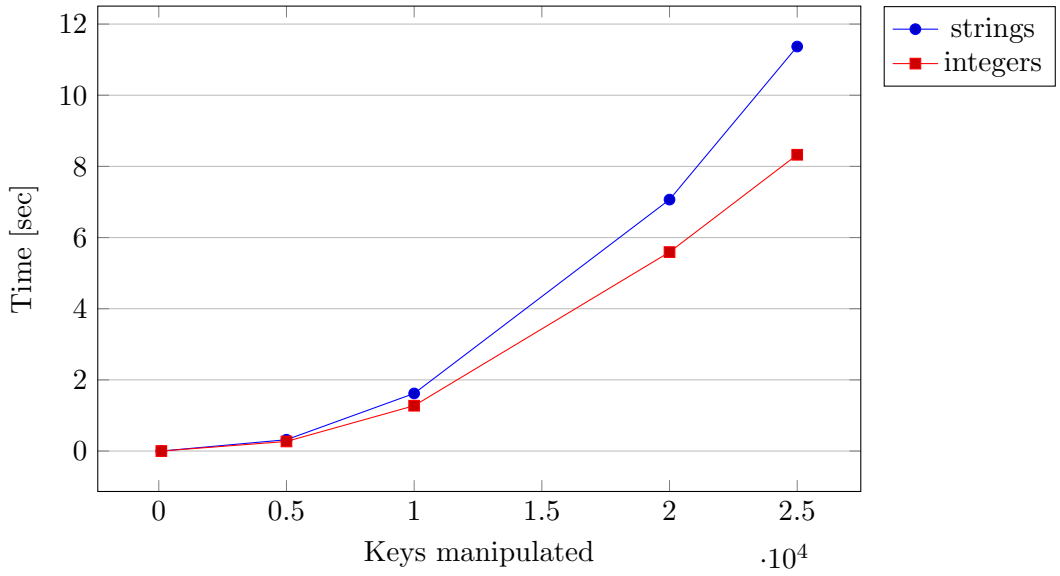


Figure 3.3: Test with 75% insertions, 25% deletions

The objective of the second test (3.3) is to display more casual situation, where at some point during the computation, the database system is out of remaining capacity. This shows how the routine of deletion, when performed, can affect the total performance.

In this case, we see the similar pattern, than in the first test (3.2). The reason being is, that the deletion routine that is performed in this set of operations is as same as insertion, largely consisting of the search function. For both of these procedures, first, the right position (element) has to be isolated.

At the level of the data component, the overall slowness of this test when compared to the first test (3.2) is caused by two factors. Firstly, this operation introduces rotations as well, although not in the same manner. The second, more important is, that when deleting the node from the data component (AVL tree) which is having both children, there is an operation of finding a successor of the node deleted for replacement. This operation can be slow, especially, when deleting the root node. That is why, in the parallel discussion, by keeping the current successor and predecessor of the nodes we help to speed up the algorithm.

When considering the priority component, removal of a node does invoke the

bubble-down repair function. When the removal happens for a capacity reason and is not executed by the command, the deletion removes the top element of the (minimal) binary heap, the one having the smallest priority and leading to the worst case of removal in terms of time.

3.5.4 Test with 40% insertions, 10% deletions and 50% retrievals

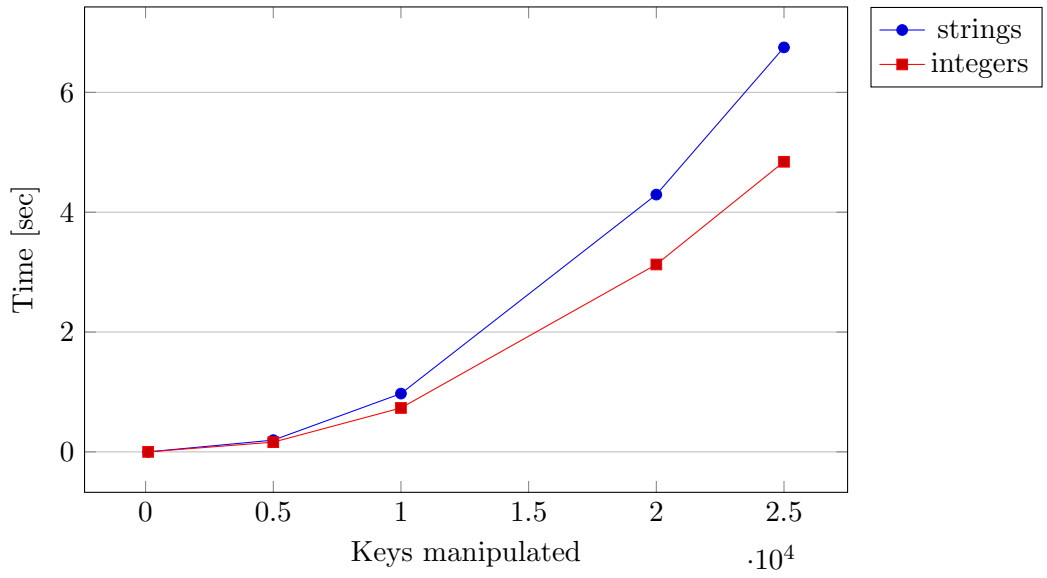


Figure 3.4: Test with 40% insertions, 10% deletions and 50% retrievals

In this third test, the most realistic scenario is measured. The numerical database system was primarly optimised for search (retrieve) function. In this test, the search operation does takes place the most.

Notably, time of keys being manipulated did go down and in this configuration we can effectively manipulate with 1000 keys with 40-10-50 operation split in under a second.

3.5.5 Comparison to the initial implementation

One part of the thesis assignment was, to compare the performance of my numerical database system, to the original implementation [1]. Although the maximal efforts to do so, I was unable to run and evaluate such comparison. The reason being was not the well-documented use of the old system and missing guide to compile and subsequently, execute the DRIVER [1] routine.

An important note is, that my tests do seem to copy trends of the performance of the original implementation. This is assumed while looking at the graph provided in the original paper [1]. The graph can be seen in the figure below 3.5.

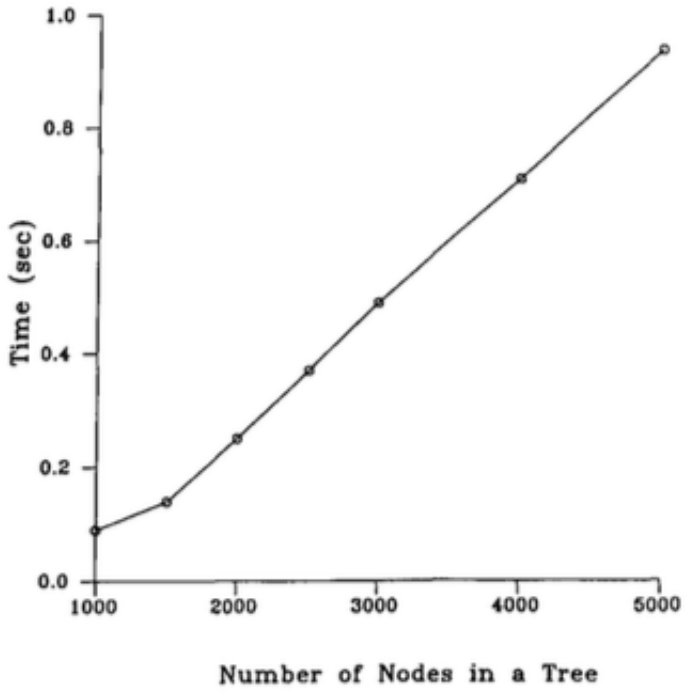


Figure 3.5: Performance of the original implementation [1]

The methodology of testing the original implementation is the average time for fetch, insert and delete operations on a sample database (database is created using simple random generators). These are then plotted as a function of a number of nodes in the database system. Tests were performed on the mainframe computer IBM 3090/600J [1].

Conclusion

In conclusion, I would like to comment on this thesis by saying, that the objective of the thesis was fulfilled. The numerical database system was thoroughly studied and explained in great detail. I provided purposes, expectations on which of operations shall be supported and their corresponding asymptotic notations. I introduced the different interpretation of data structures compared to original implementation. I made the system modular, and ready for further developments. I did study materials of parallel processing of tree structures, aligned and prepared the model that will support an implementation of these changes. I implemented the numerical database system in C++ language, using OOP paradigm and presented the interface of classes implemented. Finally, I did measure and show how my implementation differs from the initial one. I could not perform the exact measurement comparison, so I provided graphs showing the trend of performance between my implementation and the original one.

Future prospects

As I enjoyed the study and implementation of this database system, I created a good foundation for future studies and improvements of this concept. In the future, other scholars or really anyone interested in the topic, myself included, can experiment and extend this system to offer more functionality or to implement all parallel suggestions provided in this thesis.

I would like to continue in such efforts in my Master thesis if possible.

Bibliography

[1] Park, S. C.; Bahri, C.; Draayer, J. P.; et al. Numerical database system based on a weighted search tree. *Computer Physics Communications*, volume 82, no. 2-3, 1994: pp. 247–264.

[2] Beynon–Davies, P. *Database Systems*. Boston: Palgrave Macmillan, third edition, 2003, ISBN 978-1403916013.

[3] Ramakrishnan, R.; Gehrke, J. *Database Systems*. Boston: McGraw-Hill Companies, second edition, 2000, ISBN 978-0072465358.

[4] Sedgewick, R. *Algorithms In C:*. 1-4, Boston: Pearson Education., third edition, 1998, ISBN 978-81-317-1291-7.

[5] Garnier, R.; Taylor, J. *Discrete Mathematics:*. Miami: CRC Press, third edition, 2009, ISBN 978-1-4398-1280-8.

[6] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; et al. *Introduction to Algorithms*. Cambridge: MIT Press and McGraw-Hill, third edition, 2009, ISBN 0-262-03384-4.

[7] Aho, A. V.; Hopcroft, J. E.; Ullman, J. D. *The Design and Analysis of Computer Algorithms*. Cambridge: Addison-Wesley, 1974, ISBN 978-0201000290, 145-147 pp.

[8] Sleator, D. D.; Tarjan, R. E. Self-Adjusting Binary Search Trees. *Journal of the ACM (Association for Computing Machinery)*, volume 32, no. 3, 1985: p. 652–686.

[9] Heger, D. A. A Disquisition on The Performance Behaviour of Binary Search Tree Data Structures. 2016, [Online; accessed 2016-04-01]. Available from: `http://www.cepis.org/upgrade/files/full-2004-V.pdf`

[10] Knuth, D. *The Art of Computer Programming*. Boston: Addison-Wesley, third edition, 1997, ISBN 0-201-89685-0, 458–475 pp.

[11] AVL Trees. 2016, [Online; accessed 2016-04-01]. Available from: http://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html

[12] Lecture 11: Red-Black Trees. 2016, [Online; accessed 2016-04-01]. Available from: http://web.eecs.umich.edu/~sugih/courses/eecs281/f11/lectures/11-Redblack.pdf

[13] Wikipedia. Tree (data structure) — Wikipedia, The Free Encyclopedia. 2016, [Online; accessed 2016-04-01]. Available from: https://en.wikipedia.org/wiki/Tree_(data_structure)

[14] Wikipedia. Binary tree — Wikipedia, The Free Encyclopedia. 2016, [Online; accessed 2016-04-01]. Available from: https://en.wikipedia.org/wiki/Binary_tree

[15] 6.1 Binary Heaps. 2016, [Online; accessed 2016-04-01]. Available from: http://lcm.csa.iisc.ernet.in/dsa/node137.html

[16] Drachsler, D.; Vechev, M.; Yahav, E. Practical Concurrent Binary Search Trees via Logical Ordering. 2014, [Online; accessed 2016-04-01]. Available from: http://www.cs.technion.ac.il/~yahave/papers/ppopp14-trees.pdf

[17] Bronson, N. G.; Casper, J.; Chafi, H.; et al. A Practical Concurrent Binary Search Tree. 2009, [Online; accessed 2016-04-01]. Available from: http://arsenalfc.stanford.edu/papers/ppopp207-bronson.pdf

[18] Stroustrup, B. Lecture:The essence of C++. University of Edinburgh. [Online; accessed 2016-04-01]. Available from: https://www.youtube.com/watch?v=86xWVb4XIyE

[19] Google Inc. *Womens World Cup Players.* [Online; accessed 2016-03-15]. Available from: https://github.com/GoogleTrends/data/blob/gh-pages/20150512_WomensWorldCupPlayers.csv

[20] Backus, J. The history of FORTRAN I, II and III. 2016, [Online; accessed 2016-04-01]. Available from: http://www.softwarepreservation.org/projects/FORTRAN/paper/p25-backus.pdf

[21] String vs. int comparison which is faster? 2011, [Online; accessed 2016-04-01]. Available from: http://www.gamedev.net/topic/598968-string-vs-int-comparison-which-is-faster/

[22] Holte, R. C. 4. Implementing a Tree in an Array. [Online; accessed 2016-04-01]. Available from: https://webdocs.cs.ualberta.ca/~holte/T26/tree-as-array.html

[23] Williams, A. *C++ Concurrency in Action*. Cambridge: Manning Publications, 2012, ISBN 860-1200915495.

[24] Zheng, S. Q. A simple and powerful representation of binary search trees. *Lecture Notes in Computer Science*, volume 507, 2005: pp. 192–198.

[25] GNU GCC. *3.10 Options That Control Optimization*. [Online; accessed 2016-05-01]. Available from: `https://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_3.html`

Appendix A

Definitions

Definition A.1 *DBMS stands for "Database Management System." In short, a DBMS is a database program.*

Christensson, P. *DBMS Definition* [Online; accessed 2016-04-01]. Available from: `http://techterms.com`

Definition A.2 *Readers–writer lock (RWL) or shared-exclusive lock (also known as a multiple readers/single-writer lock or multi-reader lock) is a synchronization primitive that solves one of the readers–writers problems. An RW lock allows concurrent access for read-only operations, while write operations require exclusive access.*

Hamilton, Doug *Readers–writer lock* [Online; accessed 2016-04-01]. Available from: `https://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock`

Definition A.3 *A procedure is thread safe when the procedure is logically correct when executed simultaneously by several threads.*

Oracle Corporation *Thread safe procedure* [Online; accessed 2016-04-01]. Available from: `http://download.oracle.com/docs/cd/E19963-01/html/821-1601/docinfo.html`

Definition A.4 *The LLVM compiler infrastructure project (formerly Low Level Virtual Machine) is a "collection of modular and reusable compiler and toolchain technologies" used to develop compiler front ends and back ends.*

The LLVM Foundation *LLVM compiler* [Online; accessed 2016-04-01]. Available from: `http://llvm.org/`

Appendix B

Acronyms

DBMS Database Management System

C++ General-purpose Programming Language

BT Binary Tree

BST Binary Search Tree

BBST Balanced Binary Search Tree

AVL Self BBST named after Georgy Adelson-Velsky and Evgenii Landis'

BH Binary Heap

OOP Object Oriented Programming

STL Standard Template Library

CPU Central Processing Unit

RAM Random Access Memory

HDD Harddisk Drive

OS Operating System

Appendix C

Attachments

```
template <class T>
class CTerm {
public:
    T key;
    int balance;
    CTerm *left, *right, *parent;

    CTerm(T k, CTerm *p);
    ~CTerm();
};
```

Figure C.1: CTerm.h

```
template <class T>
class CPriority {
public:
    T key;
    float priority;

    CPriority(T k, float p);
    ~CPriority();
};
```

Figure C.2: CPriority.h

Appendix D

Contents of enclosed SD-card

```
Makefile ..... set of directives to compile the database system
main.cpp ..... main routine for demonstration of database system
src ..... the directory of source codes
    CDatabase.cpp ..... implementation of CDatabase wrapper
    CDatabase.h ..... header of CDatabase wrapper
    CTerm.cpp ..... implementation of a node of the data component
    CTerm.h ..... header of a node of the data component
    CTermTree.cpp ... implementation of AVL tree of the data component
    CTermTree.h ..... header of AVL tree of the data component
    CPriority.cpp .. implementation of a node of the priority component
    CPriority.h ..... header of a node of the priority component
    CPriorityHeap.cpp . implementation of BH of the priority component
    CPriorityHeap.h ..... header of BH of the priority component
text ..... the thesis text directory
    thesis.pdf ..... the thesis text in PDF format
    thesis.tex ..... the thesis text in latex format
data ..... sample of data for testing purposes
    sample.txt ..... main dataset for testing
    sampleInt.txt ..... integer dataset for testing
    sampleString.txt ..... string dataset for testing
readme.txt ... the file with SD-card contents description and instructions
licence.txt ..... licence to the code for this thesis
```